1. vector (มาตรฐาน C++11)

vector เป็นหนึ่งใน C++ template class ในกลุ่มของ container จาก Standard Template Library (STL) และ vector เป็นส่วนหนึ่งของ STL ตั้งแต่มาตรฐานแรกของ C++ (C++98) เช่นเดียวกับ list, stack, queue, map, unorder_map, set และ priority_queue

vector เป็น container ที่เก็บข้อมูลทั้งหมดไว้ในหน่วยความจำที่มีตำแหน่งต่อเนื่องกันเช่นเดียวกับอาร์เรย์ในทางปฏิบัติแล้ว vector สร้างมาจากอาร์เรย์นั้นเอง เพียงแต่ vector เป็นอาร์เรย์แบบที่เรียกว่า dynamic array ซึ่งก็คืออาร์เรย์ที่ไม่จำเป็นต้องกำหนดขนาดเมื่อแรกสร้าง ขนาดของอาร์เรย์จะเพิ่มขึ้นและลดลงไปตามจำนวนข้อมูลที่อยู่ในอาร์เรย์อัตโนมัติ

ในด้านการพัฒนาแล้ว vector จะสร้างอาร์เรย์ที่กำหนดขนาดเริ่มต้นไว้ (capacity) หากเมื่อมีการบรรจุข้อมูลลงในอาร์เรย์จนเกินกว่าที่ขนาดของอาร์เรย์จะรองรับได้ จะมีการสร้างอาร์เรย์ที่ใหญ่กว่าเดิม (reallocation) จากนั้นทำการคัดลอกข้อมูลจากอาร์เรย์เดิมไปใส่ในอาร์เรย์ใหม่ รวมถึงนำข้อมูลใหม่ที่ไม่สามารถบรรจุในอาร์เรย์เดิมได้ ไปใส่ในอาร์เรย์ใหม่ วิธีการพัฒนาแบบนี้เป็นที่มาของจุดแข็งและจุดอ่อนของ vector โดยเฉพาะเมื่อเทียบกับ list (ที่พัฒนามาจาก linked list)

การเรียกใช้งาน vector ต้องใช้คำสั่ง preprocessor directive `#include<vector>`

การประกาศ vector ใช้รูปแบบต่อไปนี้

- `vector<type> vec` เพื่อสร้าง vector ที่ไม่มีข้อมูลใด ๆ (empty vector) ซึ่งมีขนาดเป็น 0 (Time Complexity $\theta(1)$)
- `vector<type> vec(N)` เพื่อสร้าง vector ขนาด `N` (Time Complexity: $\theta(N)$)

การประกาศและการกำหนดค่าเริ่มต้นให้ vector ใช้รูปแบบต่อไปนี้

- `vector<type> vec(N,a)` เพื่อสร้าง vector ขนาด `N` ที่บรรจุข้อมูล `a` ในทุกตำแหน่ง โดยที่ข้อมูล `a` ควรเป็นชนิด `T` (ถ้าไม่เป็น อาจเกิด compilation error หรือ implicit type conversion) (Time Complexity: $\theta(N)$)

- `vector<type> vec(vec1)` เพื่อสร้าง vector ใหม่ที่คัดลอกข้อมูลมาจาก vector เดิมที่ชื่อ `vec1` โดยที่ `vec` มีขนาดเท่ากับ `vec1` (Time Complexity: $\theta(\max(N,M))$ เมื่อ N,M เป็นขนาดของ `vec` และ `vec1` ตามลำดับ)

ตัวดำเนินการของ vector (`N` คือขนาดของ vector)

Operator	Description	Time Complexity
`Operator[i]`	Returns <u>a reference</u> to the element at position `i` in the vector.	$\theta(1)$
`Operator=`	Assigns new content to the vector, replacing its current contents, resize to fit new content.	$O(\max(N,M))$ เมื่อ N,M เป็นขนาดของ vector

ฟังก์ชันของ vector (`N` คือขนาดของ vector `v`)

Function	Description	Time Complexity
`v.size()`	Returns the number of elements in the vector.	$\theta(1)$
`v.max_size()`	Returns the maximum number of elements that the vector can hold.	$\theta(1)$
`v.resize(M)` `v.resize(M, a)`	Resizes the vector so that it contains `M` elements. Resizes the vector so that it contains `M` elements, if the new size is greater than the old one, fill all new positions with `a`.	$\theta(\|N-M\|)$
`v.capacity()`	Returns the size of the storage space currently allocated for the vector (`vec.size() <= vec.capacity()`)	$\theta(1)$
`v.empty()`	Returns `true` if the vector size is 0, `false` otherwise.	$\theta(1)$

Function	Description	Time Complexity
`v.reserve(M)`	Requests that the vector capacity be at least enough to contain `M` elements.	$O(M)$ กรณี reallocation
`v.shrink_to_fit()`	Requests the vector to reduce its capacity to fit its size.	$O(N)$
`v.at(i)`	Returns a reference to the element at position `i` in the vector.	$\theta(1)$
`v.front()`	Returns a reference to the first element in the vector.	$\theta(1)$
`v.back()`	Returns a reference to the last element in the vector	$\theta(1)$
`v.data()`	Returns a direct pointer to the memory array used internally by the vector.	$\theta(1)$
`v.assign(M, a)`	Assigns new contents of all `a` to the vector, replacing its current contents, resize to size `M`.	$O(\max(N,M))$
`v.push_back(a)`	Adds a new element at the end of the vector. The content of `a` is copied (or moved) to the new element.	$\theta(1)$ หรือ $O(N)$ กรณี reallocation

`v.insert(it, a)`
 	
`v.insert(it,n,a)`
	The vector is extended by inserting a new element `a` before the element pointed by iterator `it`.
	The vector is extended by inserting `n` new elements `a` before the element pointed by iterator `it`.
		$O(n+N)$

2. หลักการนับเบื้องต้น การเรียงสับเปลี่ยน และการจัดหมู่

หลักการนับเบื้องต้น

หลักการนับเบื้องต้นประกอบด้วยกฎ 4 กฎคือคือ กฎการคูณ กฎการบวก กฎการลบ และกฎการหาร ในบทเรียนนี้จะกล่าวถึงเฉพาะ 3 กฎแรก

กฎการคูณ (The Product Rule)

สมมุติว่า มีกระบวนงานที่สามารถถูกแบ่งออกเป็นงาน k ชิ้นที่ต้องทำทั้งหมด

ถ้า มี n_1 วิธีในการทำงานชิ้นที่ 1 และ

สำหรับแต่ละวิธีของการทำงานชิ้นที่ 1 นั้นมี n_2 วิธีในการทำงานชิ้นที่ 2 และ

สำหรับแต่ละวิธีของการทำงานชิ้นที่ 2 นั้นมี n_3 วิธีในการทำงานชิ้นที่ 3 และ

...

สำหรับแต่ละวิธีของการทำงานชิ้นที่ $k-1$ นั้นมี n_k วิธีในการทำงานชิ้นที่ k

แล้ว มี $n_1 \times n_2 \times n_3 \times \ldots \times n_k$ วิธีในการทำงานทั้งกระบวนงาน

ตัวอย่างที่ 2.1 จงหาจำนวนบิทสตริง (bit string) ความยาว 8 ตัวอักขระ

คำตอบ ใช้กฎการคูณ โดยการแบ่งกระบวนงานออกเป็น 8 งาน แต่ละงานคือการกำหนดค่าของแต่ละตำแหน่งของบิทสตริง ซึ่งแต่ละตำแหน่งสามารถกำหนดค่าได้ 2 แบบคือ 0 หรือ 1

ดังนั้นจำนวนของบิทสตริงความยาว 8 ตัวอักขระคือ $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$

ตัวอย่างที่ 2.2 จงหาจำนวนบิทสตริง (bit string) ความยาว 8 ตัวอักขระที่เริ่มต้นด้วย 1 และลงท้ายด้วย 00

คำตอบ ใช้กฎการคูณ โดยการแบ่งกระบวนงานออกเป็น 8 งาน

ตำแหน่ง	1	2	3	4	5	6	7	8
	1	?	?	?	?	?	0	0

งานที่ 1 ทำได้ 1 วิธี คือต้องกำหนดให้ตำแหน่งที่ 1 มีค่าเป็น 1

งานที่ 2-6 แต่ละงานทำได้ 2 วิธี คือเลือก 0 หรือ 1

งานที่ 7 ทำได้ 1 วิธี คือต้องกำหนดให้ตำแหน่งที่ 7 มีค่าเป็น 0

งานที่ 8 ทำได้ 1 วิธี คือต้องกำหนดให้ตำแหน่งที่ 8 มีค่าเป็น 0

ดังนั้นจำนวนของบิทสตริงความยาว 8 ตัวอักขระที่เริ่มต้นด้วย 1 และลงท้ายด้วย 00 คือ $1\times 2^5 \times 1\times 1 = 32$

ตัวอย่างที่ 2.3 กำหนดให้ user name ของอีเมลล์ภายใต้ชื่อโดเมน @pattani.psu.ac.th ต้องใช้ตัวอักขระ (character) ที่เป็น (1) ตัวอักขระภาษาอังกฤษตัวพิมพ์เล็ก (2) ตัวพิมพ์ใหญ่ หรือ (3) ตัวเลข 0-9 โดยตัวอักขระตัวพิมพ์เล็กหรือพิมพ์ใหญ่มีความต่าง ตัวอย่างเช่น ที่อยู่ ratta@pattani.psu.ac.th ไม่เป็นที่อยู่เดียวกับ Ratta@pattani.psu.ac.th จงหาจำนวนของ user name ของอีเมลล์ภายใต้ชื่อโดเมน @pattani.psu.ac.th ที่มีความยาว 5 ตัวอักขระ

คำตอบ ใช้กฎการคูณโดยแบ่งกระบวนงานเป็น 5 งานตามตำแหน่งของตัวอักขระ

งานที่ 1 คือการเลือกว่าจะกำหนดให้ตำแหน่งที่ 1 เป็นตัวอักขระอะไร ซึ่งจะทำได้ 62 วิธี เพราะเลือกจากสมาชิกของเซต $\{A,B,C,...,Z\}\cup\{a,b,c,...,z\}\cup\{0,1,2,3,...,9\}$

งานที่ 2 เช่นเดียวกับงานที่ 1 จึงทำได้ 62 วิธี

งานที่ 3 เช่นเดียวกับงานที่ 1 จึงทำได้ 62 วิธี

งานที่ 4 เช่นเดียวกับงานที่ 1 จึงทำได้ 62 วิธี

งานที่ 5 เช่นเดียวกับงานที่ 1 จึงทำได้ 62 วิธี

ใช้กฎการคูณ มีจำนวน user name ทั้งหมด $62\times 62\times 62\times 62\times 62 = 62^5 = 916,132,832$ แบบ

กฎการบวก (The Sum Rule)

ถ้า งานชิ้นหนึ่งสามารถถูกทำด้วย วิธีที่ 1 หรือ วิธีที่ 2 โดยที่

วิธีที่ 1 มี n_1 ทางเลือกในการเลือกทำ และ วิธีที่ 2 มี n_2 ทางเลือกในการเลือกทำ

และไม่มีทางเลือกใดใน n_1 ทางเลือกที่ซ้ำกับทางเลือกใน n_2 **แล้ว** มี $n_1 + n_2$ ทางเลือกในการทำงานชิ้นนั้น

ให้ A_1 เป็นเซตของทางเลือกในการทำงานด้วยวิธีที่ 1 ดังนั้น $|A_1| = n_1$

A_2 เป็นเซตของทางเลือกในการทำงานด้วยวิธีที่ 2 ดังนั้น $|A_2| = n_2$

สำหรับเงื่อนไขของกฎการบวกแล้ว $|A_1 \cap A_2| = \varnothing$ ทำให้ได้ว่า $|A_1 \cup A_2| = |A_1| + |A_2| = n_1 + n_2$

ตัวอย่างที่ 2.4 จงหาจำนวนบิทสตริง (bit string) ความยาว 8 หรือความยาว 9 ตัวอักขระ

คำตอบ ใช้กฎการคูณและกฎการบวก

ใช้กฎการบวกในการพิจารณาการสร้างบิทสตริง 2 วิธี

วิธีที่ 1 สร้างบิทสตริงความยาว 8 ตัวอักขระ ใช้กฎการคูณหาจำนวนบิทสตริงความยาว 8 ตัวอักขระได้ 2^8 แบบ

วิธีที่ 2 สร้างบิทสตริงความยาว 9 ตัวอักขระ ใช้กฎการคูณหาจำนวนบิทสตริงความยาว 9 ตัวอักขระได้ 2^9 แบบ

ข้อสังเกตไม่มีบิทสตริงความยาว 8 ตัวอักขระที่ซ้ำกับบิทสตริงความยาว 9 ตัวอักขระ ทำให้ใช้กฎการบวกได้

ดังนั้นจำนวนบิทสตริงความยาว 8 หรือความยาว 9 ตัวอักขระคือ $2^8 + 2^9 = 768$

ตัวอย่างที่ 2.5 กำหนดให้ password ต้องมีความยาว 5 ตัว โดยที่แต่ละตัวสามารถเป็นตัวอักขระภาษาอังกฤษตัวพิมพ์ใหญ่ใด ๆ (ตัวพิมพ์เล็กใช้ไม่ได้) หรือตัวอักขระพิเศษใด ๆ จากเซต {$, %, &, #, *} มี password กี่รูปแบบที่มีตัวอักขระพิเศษจากเซต {$, %, &, #, *} อย่างน้อย 1 ตัวใน password นั้น

คำตอบ ใช้กฎการคูณและกฎการบวก

สังเกตว่าเซตของ password ความยาว 5 ตัวอักขระสามารถแบ่งได้เป็น 2 เซตคือ

1) เซตของ password ความยาว 5 ตัวอักขระที่ไม่มีตัวอักขระพิเศษเลย (มีตัวอักขระพิเศษ 0 ตัว)
2) เซตของ password ความยาว 5 ตัวอักขระที่มีตัวอักขระพิเศษอย่างน้อย 1 ตัว (มีตัวอักขระพิเศษ 1 ถึง 5 ตัว)

กำหนดให้ A เป็นเซตของ password ความยาว 5 ตัวอักขระ (ทุกแบบ)

B เป็นเซตของ password ความยาว 5 ตัวอักขระที่ไม่มีตัวอักขระพิเศษเลย (มีตัวอักขระพิเศษ 0 ตัว)

C เป็นเซตของ password ความยาว 5 ตัวอักขระที่มีตัวอักขระพิเศษอย่างน้อย 1 ตัว

ดังนั้น $A = B \cup C, B \cap C = \varnothing$ ตรงตามเงื่อนไขของกฎการบวก ทำให้ได้ว่า $|A| = |B| + |C|$

หาค่าของ $|C|$ ซึ่งสามารถหาได้จาก $|C| = |A| - |B|$

หาค่าของ $|A|$ ซึ่งคือจำนวน password ความยาว 5 ตัวอักขระ (ทุกแบบ)

ใช้กฎการคูณโดยแบ่งกระบวนงานออกเป็น 5 งานตามตำแหน่งของตัวอักขระ

งานที่ 1 คือการเลือกว่าจะกำหนดให้ตำแหน่งที่ 1 เป็นตัวอักขระอะไร ซึ่งจะทำได้ 31 วิธี เพราะเลือกจากเซต $\{A, B, C, ..., Z\} \cup \{\$, \%, \&, \#, *\}$

งานที่ 2 - งานที่ 5 แต่ละงานเหมือนงานที่ 1

ใช้กฎการคูณ ทำให้ได้ $|A| = 31^5 = 28{,}629{,}151$ แบบ

หาค่าของ $|B|$ ซึ่งคือการหาจำนวน password ความยาว 5 ตัวอักขระไม่มีตัวอักขระพิเศษเลย (มีตัวอักขระพิเศษเป็นจำนวน 0 ตัว)

ใช้กฎการคูณโดยแบ่งกระบวนงานเป็น 5 งานตามตำแหน่งของตัวอักขระ

งานที่ 1 คือการเลือกว่าจะกำหนดให้ตำแหน่งที่ 1 เป็นตัวอักขระอะไร ซึ่งจะทำได้ 26 วิธี เพราะเลือกจากเซต $\{A, B, C, ..., Z\}$

งานที่ 2 - งานที่ 5 แต่ละงานเหมือนงานที่ 1

ใช้กฎการคูณ ทำให้ได้ $|B| = 26^5 = 11,881,376$ แบบ

$$
\begin{aligned}
|C| &= |A| - |B| \\
&= 28,629,151 - 11,881,376 \\
&= 16,747,775
\end{aligned}
$$

ดังนั้นมี password 16,747,775 แบบที่มีตัวอักขระพิเศษอย่างน้อย 1 ตัว

กฎการลบ (The Subtraction Rule หรือ Principle of Inclusion-Exclusion)

ถ้า งานชิ้นหนึ่งสามารถถูกทำด้วย วิธีที่ 1 หรือ วิธีที่ 2 โดยที่

วิธีที่ 1 มี n_1 ทางเลือกในการเลือกทำ และ วิธีที่ 2 มี n_2 ทางเลือกในการเลือกทำ

แล้ว มี $n_1 + n_2 - n_3$ ทางเลือกในการทำงานชิ้นนั้น เมื่อ n_3 เป็นจำนวนทางเลือกที่ซ้ำกันระหว่างการทำด้วยวิธีที่ 1 และวิธีที่ 2

ให้ A_1 เป็นเซตของทางเลือกในการทำงานด้วยวิธีที่ 1 ดังนั้น $|A_1| = n_1$

A_2 เป็นเซตของทางเลือกในการทำงานด้วยวิธีที่ 2 ดังนั้น $|A_2| = n_2$

ทำให้ได้ว่า $|A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2| = n_1 + n_2 - n_3$

ตัวอย่างที่ 2.6 จงหาจำนวนบิทสตริง (bit string) ความยาว 8 ตัวอักขระที่เริ่มต้นด้วย 1 หรือลงท้ายด้วย 00

คำตอบ ใช้กฎการลบร่วมกับกฎการคูณ

กำหนดให้ A เป็นเซตของบิทสตริงความยาว 8 ตัวอักขระที่เริ่มต้นด้วย 1 หรือลงท้ายด้วย 00

B เป็นเซตของบิทสตริงความยาว 8 ตัวอักขระที่เริ่มต้นด้วย 1

C เป็นเซตของบิทสตริงความยาว 8 ตัวอักขระที่ลงท้ายด้วย 00

ดังนั้น $B \cap C$ จึงเป็นเป็นเซตของบิทสตริงความยาว 8 ตัวอักขระที่เริ่มต้นด้วย 1 และลงท้ายด้วย 00

ใช้กฎการคูณหา $|B|=2^7$, $|C|=2^6$ และ $|B\cap C|=2^5$

จากกฎการลบ $|A|=|B|+|C|-|B\cap C|=128+64-32=160$

ดังนั้นจำนวนของบิทสตริงความยาว 8 ตัวอักขระที่เริ่มต้นด้วย 1 <u>หรือ</u>ลงท้ายด้วย 00 คือ 160

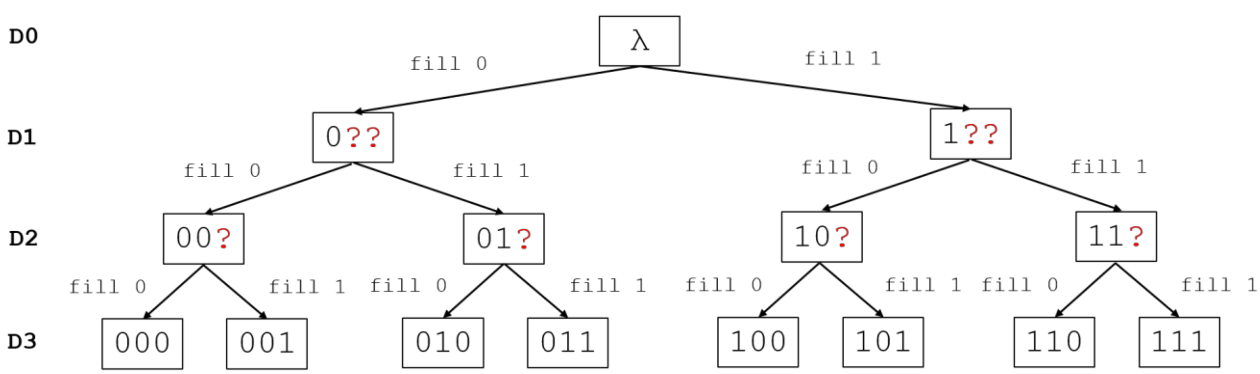
การสร้างบิทสตริงความยาว N ทุกแบบ

ขั้นตอนวิธีในการสร้างบิทสตริงความยาว N ทั้งหมด 2^N แบบได้นำแนวคิดมาจากกฎการคูณ ซึ่งก็คือการแบ่งกระบวนงานออกเป็น N งาน แต่ละงานคือการเลือก 0 หรือ 1 มาเติมในแต่ละตำแหน่งในสตริง เทคนิคการเขียนโปรแกรมของขั้นตอนวิธีนี้คือ recursion โดยการเขียน recursive function ที่มี dept ของ recursive call เป็น $N+1$ (เรียกแต่ละ dept ว่า D0, D1, ..., DN ตามลำดับ)

เมื่อเริ่มต้นการทำงาน recursive function ถูกเรียกด้วย empty string (λ) ที่ dept D0 จากนั้นขั้นตอนวิธีจะทำ 2 อย่างคือ

(1) เติมตำแหน่งที่ 1 ด้วย 0 และเรียก recursive function ที่ dept D1 เพื่อให้เติมค่าที่ตำแหน่งที่ 2 ถึง N
(2) เติมตำแหน่งที่ 1 ด้วย 1 และเรียก recursive function ที่ dept D1 เพื่อให้เติมค่าที่ตำแหน่งที่ 2 ถึง N

การเติมค่าที่ตำแหน่งที่ 2 จะถูกจัดการด้วย recursive call ที่ dept D1 การเติมค่าที่ตำแหน่งที่ 3 จะถูกจัดการด้วย recursive call ที่ dept D2 ทำเช่นนี้จนถึงที่ dept DN ซึ่งจะพบ base case ทำให้ไม่มีการเรียก recursive call อีกต่อไป แต่คืนสตริงที่เป็นผลลัพธ์ออกมา



รูปที่ 2.1

รูปที่ 2.1 แสดงตัวอย่างของการทำงานของขั้นตอนวิธีเพื่อสร้างบิทสตริงความยาว 3 ตัวอักขระ recursive call มี dept เท่ากับ 4 เรียกแต่ละ dept ว่า D0, D1, D2 และ D3

- ที่ D0 ขั้นตอนวิธีจะเติมตำแหน่งที่ 1 ด้วย 0 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 0?? ใน D1
- ที่ D0 ขั้นตอนวิธีจะเติมตำแหน่งที่ 1 ด้วย 1 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 1?? ใน D1
- ที่ D1 ขั้นตอนวิธีจะเติมตำแหน่งที่ 2 ด้วย 0 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 00? ใน D2
- ที่ D1 ขั้นตอนวิธีจะเติมตำแหน่งที่ 2 ด้วย 1 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 01? ใน D2
- ที่ D1 ขั้นตอนวิธีจะเติมตำแหน่งที่ 2 ด้วย 0 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 10? ใน D2
- ที่ D1 ขั้นตอนวิธีจะเติมตำแหน่งที่ 2 ด้วย 1 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 11? ใน D2
- ที่ D2 ขั้นตอนวิธีจะเติมตำแหน่งที่ 3 ด้วย 0 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 000 ใน D3
- ที่ D2 ขั้นตอนวิธีจะเติมตำแหน่งที่ 3 ด้วย 1 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 001 ใน D3
- ที่ D2 ขั้นตอนวิธีจะเติมตำแหน่งที่ 3 ด้วย 0 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 010 ใน D3
- ที่ D2 ขั้นตอนวิธีจะเติมตำแหน่งที่ 3 ด้วย 1 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 011 ใน D3
- ที่ D2 ขั้นตอนวิธีจะเติมตำแหน่งที่ 3 ด้วย 0 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 100 ใน D3
- ที่ D2 ขั้นตอนวิธีจะเติมตำแหน่งที่ 3 ด้วย 1 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 101 ใน D3
- ที่ D2 ขั้นตอนวิธีจะเติมตำแหน่งที่ 3 ด้วย 0 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 110 ใน D3
- ที่ D2 ขั้นตอนวิธีจะเติมตำแหน่งที่ 3 ด้วย 1 จากนั้นทำการเรียก recursive call กับผลลัพธ์ 111 ใน D3
- ที่ D3 ตำแหน่งที่ต้องเติมคือตำแหน่งที่ 4 ซึ่งเกินจำนวนอักขระของสตริง ขั้นตอนวิธีจะพบ base case ทำให้หยุดการเรียก recursive call และคืนผลลัพธ์

ขั้นตอนวิธีนี้มี time complexity ที่ได้จาก recurrence relation $T(N) = 2T(N-1) + \theta(1)$ ซึ่งก็คือ $O(2^N)$

โปรแกรม 2.1 สร้างและแสดงผลบิทสตริงความยาว N ทุกแบบด้วยขั้นตอนวิธีที่กล่าวข้างต้น เนื่องจากจำนวนของบิทสตริงคือ 2^N โปรแกรมจึง reserve ขนาดของ vector เป็น 2^N (หรือ `1>>N`) ตั้งแต่ต้นเพื่อป้องกัน reallocation ระหว่างการเพิ่มสมาชิกให้ vector ด้วยฟังก์ชัน emplace_back( )

โปรแกรมที่ 2.1 การสร้างบิทสตริงความยาว N ตัวอักขระ (Time Complexity: $O(2^N)$)

```
// Example 2.1
#include<iostream>
#include<vector>          // vector
#include<string>          // string
using namespace std;

// Globals
vector<string> binarys;   // keeps binary strings

// Forwards
void genBinaryString(string &str, int n, int idx);
void printVector(const vector<string> &vec);

int main(){
    ios_base::sync_with_stdio(false);   // avoid syn C++ streams
    cin.tie(NULL);                      // flood cout before cin

    int N; cin >> N;
    string bString; bString.resize(N);
    binarys.reserve((1<<N));       // reserve vector capacity 2^N
    genBinaryString(bString, N, 0);
    cout << "All binary strings\n";
    printVector(binarys);
    cout << endl;
    return 0;
}
void genBinaryString(string &str, int n, int idx){
    if(idx == n)
        binarys.emplace_back(str);
    else{
        // Assigns 0 to position idx
        // fill the rest with all permutations
        str[idx] = '0';
        genBinaryString(str, n, idx + 1);

        // Assigns 1 to position idx
        // fill the rest with all permutations
        str[idx] = '1';
        genBinaryString(str, n, idx + 1);
    }
}
void printVector(const vector<string> &vec){
    for(auto &i : vec)   // O(N)
        cout<<i<< "\n";
    cout << "\n";
}
```

ผลลัพธ์

```
4
All binary strings
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

บรรทัดที่ 15 และ 16 เป็นคำสั่งที่เพิ่มความเร็วในการรับค่าด้วย cin และแสดงผลด้วย cout เพื่อให้สามารถทำงานได้เร็วเทียบเท่า scanf/printf

- `ios_base::sync_with_stdio(false)` สั่งให้โปรแกรมไม่ต้อง synchronization C++ streams กับ C streams
- `cin.tie(NULL)` สั่งให้โปรแกรม flood cout ก่อนจะรับค่าด้วย cin

การเรียงสับเปลี่ยน (Permutation)

การเรียงสับเปลี่ยน (Permutation)

การเรียงสับเปลี่ยนของเซต (ของสมาชิกที่แตกต่างกัน) คือการจัดเรียงอย่างมีลำดับกันของสมาชิกจากเซตนั้น

ตัวอย่างที่ 2.7 กำหนดเซต $S=\{a,b,c\}$

การเรียงสับเปลี่ยนของเซต S (permutation of S) มีได้ 6 รูปแบบคือ

(1) a,b,c (2) a,c,b (3) b,a,c (4) b,c,a (5) c,a,b (6) c,b,a

การเรียงสับเปลี่ยนทีละ 2 ของเซต S (2-permutation of S) มีได้ 6 รูปแบบคือ

(1) a,b (2) b,a (3) a,c (4) c,a (5) b,c (6) c,b

ทฤษฎี 1 (Theorem 1)

ถ้า n เป็นจำนวนเต็มบวก และ r เป็นจำนวนเต็มโดยที่ $1\le r\le n$

แล้ว มีจำนวนการเรียงสับเปลี่ยนทีละ r (r-permutation) ของเซต (ที่มีสมาชิกไม่ซ้ำกัน) n ตัวเท่ากับ

$$P(n,r)=n(n-1)(n-1)...(n-r+1)$$

จากทฤษฎี 1 จะได้ว่า $P(n,n)=n(n-1)(n-1)...(n-n+1)=n(n-1)(n-1)...(1)=n!$

บทเทียบ 1 (Corollary 1)

ถ้า n เป็นจำนวนเต็มบวก และ r เป็นจำนวนเต็มโดยที่ $0\le r\le n$

แล้ว $P(n,r)=\dfrac{n!}{(n-r)!}$

จากบทเทียบ 1 จะได้ว่า $P(n,0)=\dfrac{n!}{(n-0)!}=\dfrac{n!}{n!}=1$

ตัวอย่างที่ 2.8 ในการเล่นเกมจับฉลากที่มีผู้เล่น 111 แต่ละคนได้รับฉลากที่มีหมายเลข 1-111 ที่ไม่ซ้ำกันกำกับไว้ มีรางวัล 4 รางวัลคือ (1) โทรทัศน์จอแบน (2) Power bank (3) เครื่องปิ้งขนมปัง (4) บัตรของขวัญมูลค่า 1000 บาท มีกี่วิธีในการให้รางวัล ที่ผู้มีฉลากหมายเลข 100 ต้องได้รับรางวัลใดรางวัลหนึ่งใน 4 รางวัล

คำตอบ ใช้กฎการคูณและความรู้เรื่องการเรียงสับเปลี่ยน

ใช้กฎการคูณโดยแบ่งกระบวนงานออกเป็น 2 งานคือ

งานที่ 1 เลือกรางวัล 1 ใน 4 รางวัลให้กับผู้มีฉลากหมายเลข 100 ทำได้ 4 วิธี

งานที่ 2 เหลือรางวัล 3 รางวัลและเหลือผู้มีสิทธิ์ได้รับรางวัล 110 คน จำนวนวิธีในการให้รางวัล 3 รางวัลที่เหลือกับผู้มีสิทธิ์ได้รับรางวัล 110 คนก็คือ จำนวนการเรียงสับเปลี่ยนทีละ 3 ของเซตที่มีสมาชิก 110 ตัวคือ

$P(110,3) = 110 \times 109 \times 108 = 1,294,920$ วิธี

จากกฎการคูณ จำนวนวิธีในการให้รางวัลที่ผู้มีฉลากหมายเลข 100 ต้องได้รับรางวัลใดรางวัลหนึ่งใน 4 รางวัล คือ

$4 \times 1,294,920 = 5,179,680$ วิธี

ตัวอย่างที่ 2.9 ในการเล่นเกมจับฉลากที่มีผู้เล่น 111 แต่ละคนได้รับฉลากที่มีหมายเลข 1-111 ที่ไม่ซ้ำกันกำกับไว้ มีรางวัล 4 รางวัลคือ (1) โทรทัศน์จอแบน (2) Power bank (3) เครื่องปิ้งขนมปัง (4) บัตรของขวัญมูลค่า 1000 บาท มีกี่วิธีในการให้รางวัล ที่ผู้มีฉลากหมายเลข 99 และ หมายเลข 100 ต้องได้รับรางวัลทั้งคู่ แต่จะเป็นรางวัลใดก็ได้

คำตอบ ใช้กฎการคูณและความรู้เรื่องการเรียงสับเปลี่ยน

ใช้กฎการคูณโดยแบ่งกระบวนงานออกเป็น 2 งานคือ

งานที่ 1 เลือกรางวัล 2 รางวัลจาก 4 รางวัลมามอบให้ผู้มีฉลากหมายเลข 99 และผู้มีฉลากหมายเลข 100 เพื่อให้คนทั้งสองได้รับรางวัล ซึ่งทำได้เท่าจำนวนการเรียงสับเปลี่ยนทีละ 2 ของเซตที่มีสมาชิก 4 ตัวคือ

$P(4,2) = 4 \times 3 = 12$ วิธี

<u>งานที่ 2</u> เหลือรางวัล 2 รางวัลและเหลือผู้มีสิทธิ์ได้รับรางวัล 109 คน จำนวนวิธีในการให้รางวัล<u>ที่เหลือ</u> 2 รางวัลกับผู้มีสิทธิ์ได้รับรางวัล 109 คนก็คือ จำนวนการเรียงสับเปลี่ยนทีละ 2 ของเซตที่มีสมาชิก 109 ตัวคือ $P(109,2)=109\times 108=11,772$ วิธี

จากกฎการคูณ จำนวนวิธีในการให้รางวัลที่ผู้ผู้มีฉลากหมายเลข 99 และ หมายเลข 100 ต้องได้รับรางวัลทั้งคู่ คือ $12\times 11,772=141,264$ วิธี

โปรแกรมที่ 2.2 โปรแกรมคำนวณ $P(n,r)$ โดยใช้ทฤษฎี 1 (Time Complexity: $\theta(r)$)

```
#include<iostream>
using namespace std;

int main(){
    int n,r; cin >> n >> r;
    long long pnr = 1;
    for(int i=0; i<r; i++){
        pnr *= n;
        n--;
    }
    cout << pnr << endl;
    return 0;
}
```

ผลลัพธ์

```
10 5
30240
```

โปรแกรม 2.3 คำนวณ $P(n,r)$ โดยใช้บทเทียบ 1 ซึ่งจำเป็นต้องคำนวณฟังก์ชัน factorial ของเทอม $n!$ และ $(n-r)!$ อย่างไรก็ตามการเขียนฟังก์ชัน factorial แบบ recursive นั้น เมื่อ n มีค่ามาก อาจทำให้ recursive call ใช้หน่วยความจำจนหมด หรือการเขียนฟังก์ชัน factorial แบบ iterative ก็ทำให้เราต้องใช้หน่วยความจำในสำหรับอาร์เรย์ที่บันทึกค่า $k!, 0\le k\le n$ ซึ่งก็สร้างปัญหาในเรื่องหน่วยความจำเมื่อ n มีค่ามากเช่นเดียวกัน ดังนั้นเราจึงแก้ปัญหาเรื่องหน่วยความจำนี้ด้วยการใช้ฟังก์ชันคณิตศาสตร์ชื่อ Gamma ที่มีนิยามว่า $Gamma(n)=(n-1)!$ ซึ่ง STL ก็มีฟังก์ชัน Gamma ให้เรียกใช้งานได้ 2 ฟังก์ชันคือ

1. ฟังก์ชัน tgamma( ) คำนวณ $tgamma(n)=(n-1)!$
2. ฟังก์ชัน lgamma( ) คำนวณ $lgamma(n)=\ln(n-1)!$ หรือ $e^{lgamma(n)}=(n-1)!$

โปรแกรมที่ 2.3 โปรแกรมคำนวณ $P(n,r)$ โดยใช้บทเทียบ 1 และฟังก์ชัน tgamma( )

```
#include<iostream>
#include<cmath>            // tgamma()

using namespace std;
int main(){
    int n,r; cin >> n >> r;
    // tgamma(N+1) = N!
    long long pnr = ((long long) tgamma(n+1.0))/
                      ((long long) tgamma(n - r + 1.0));
    cout << pnr << endl;
    return 0;
}
```

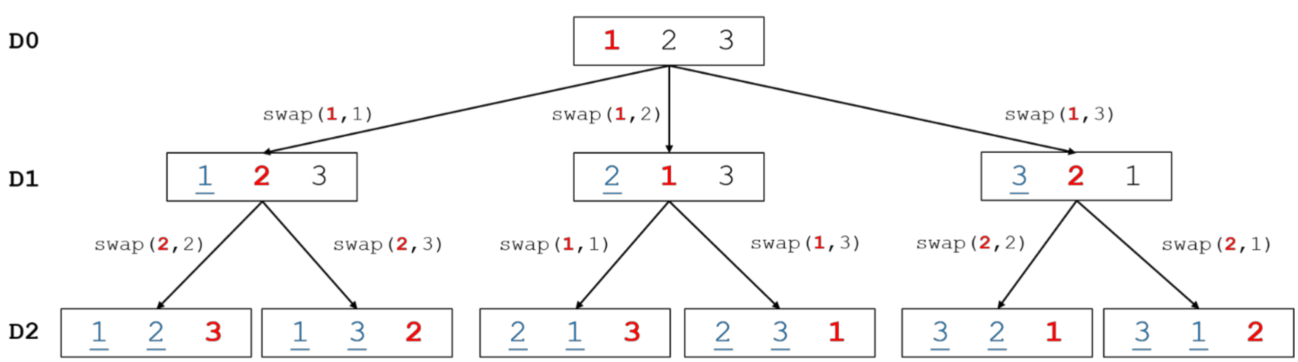
ผลลัพธ์

20 12

60339831552000

การสร้าง permutation จากสมาชิกทุกตัวในเซต

เทคนิคที่ใช้ในการเขียนโปรแกรมเพื่อสร้าง (generate) การเรียงสับเปลี่ยนของเซตจากสมาชิกทุกตัวในเซตคือ backtracking ซึ่งเป็นเทคนิคการพัฒนาขั้นตอนวิธีเพื่อการแก้ปัญหาแบบ recursive โดยการสร้างผลลัพธ์ที่ต้องการแบบ incremental



รูปที่ 2.2

รูปที่ 2.2 แสดงตัวอย่างของการทำงานของขั้นตอนวิธีเพื่อสร้างการเรียงสับเปลี่ยนทั้ง 6 รูปแบบจากเซต $S=\{1,2,3\}$ สำหรับเซตที่มีสมาชิก N ตัว dept (หรือความลึก) ของ recursive call ของขั้นตอนวิธีนี้คือ N ดังนั้น recursive call สำหรับการสร้างการเรียงสับเปลี่ยนของ $S=\{1,2,3\}$ จึงมี dept เท่ากับ 3 เรียกแต่ละ dept ว่า D0, D1 และ D2

- ที่ D0 ขั้นตอนวิธีจะทำการสลับ**สมาชิกตัวแรก**กับ (1) ตัวเอง และ (2) สมาชิกทุกตัวที่อยู่หลังสมาชิกตัวแรก เพื่อสร้างผลลัพธ์ใน D1 จากนั้นทำการเรียก recursive call กับผลลัพธ์ใน D1 ทุกตัว
- ที่ D1 ขั้นตอนวิธีจะทำการสลับ**สมาชิกตัวที่ 2** กับ (1) ตัวเอง และ (2) สมาชิกทุกตัวที่อยู่หลังสมาชิกตัวที่ 2 เพื่อสร้างผลลัพธ์ใน D2 สมาชิกที่อยู่หน้าสมาชิกตัวที่ 2 จะไม่มีการเปลี่ยนแปลง (แสดงด้วยการขีดเส้นใต้สมาชิกเหล่านั้น) จากนั้นทำการเรียก recursive call กับผลลัพธ์ใน D2 ทุกตัว
- ที่ D2 ขั้นตอนวิธีจะทำการสลับ**สมาชิกตัวที่ 3** กับตัวเองเท่านั้น (ไม่มีสมาชิกหลังสมาชิกตัวที่ 3 ให้สลับแล้ว) จากนั้นทำการเรียก recursive call อีกครั้ง ซึ่งครั้งนี้จะไม่มีสมาชิกให้สลับแล้ว ทำให้พบกับ base case ของ recursive function ที่จะคืนการเรียงสับเปลี่ยนของ D2 กลับมาเป็นผลลัพธ์สุดท้ายของขั้นตอนวิธี

สำหรับเซตที่มีสมาชิก N ตัว ขั้นตอนวิธีนี้สร้างการเรียงสับเปลี่ยน (permutation) จำนวน $N!$ แบบ และแต่ละการเรียงสับเปลี่ยนใช้เวลาในการสร้าง $\theta(N)$ (เท่ากับ dept ของ recursive call) ดังนั้น time complexity ของขั้นตอนวิธีนี้คือ $\theta(NN!)$

โปรแกรมที่ 2.4 การสร้าง permutation จากสมาชิกทุกตัวของเซตที่มีสมาชิกที่แตกต่าง

(Time Complexity: $O(NN!)$)

```
#include<iostream>
#include<vector>          // vector
#include<utility>         // swap()
#include<cmath>           // tgamma()
using namespace std;

// Globals
vector< vector <int> > perms;    // keeps permutations
// Forwards
void genPermutation(vector <int> &vec, int idx);
void printVector(const vector<int> &vec);
void print2DVector(const vector< vector <int> > &vec);

int main(){
    int N; cin >> N;
    vector<int> data;
    data.resize(N);
    perms.reserve((int) tgamma(N+1));  // tgamma(N+1) = N!
    for(auto &i : data)
        cin>>i;
    cout << "Input\n";
    printVector(data);
    genPermutation(data, 0);
    cout << "Permutation\n";
    print2DVector(perms);
    cout << endl;
    return 0;
}
void genPermutation(vector <int> &vec, int idx){
    if(idx == vec.size())
        perms.emplace_back(vec);
    else{
        for(int i=idx; i<vec.size(); i++){
            swap(vec[i], vec[idx]);
            genPermutation(vec, idx+1);
            swap(vec[i], vec[idx]);
        }
    }
}
void printVector(const vector<int> &vec){
    for(auto &i : vec)
        cout<<i<< " ";
    cout << "\n";
}
```

```
void print2DVector(const vector< vector <int> > &vec){
    for(auto &r : vec){       // O(MN)
        for(auto &c : r)
            cout << c << " ";
        cout << "\n";
    }
    cout << "\n";
}
```

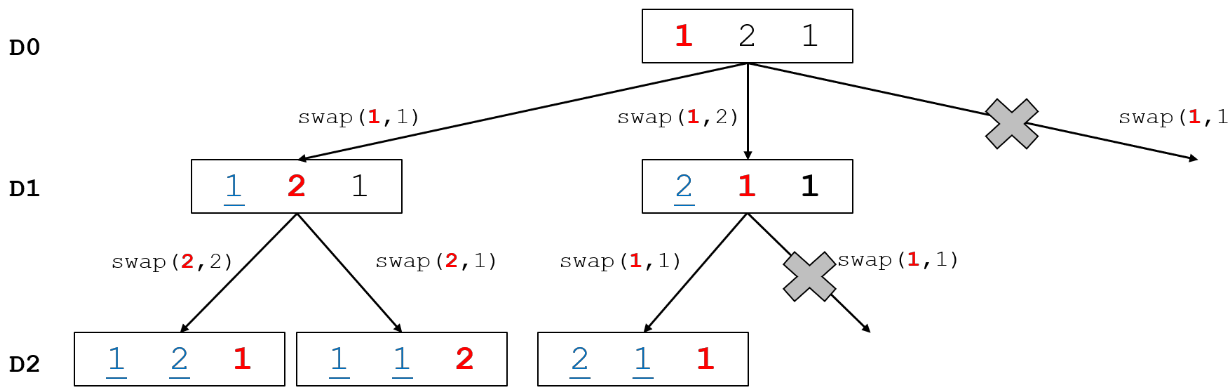
ผลลัพธ์

3

3 1 2

```
Input
3 1 2
Permutation
3 1 2
3 2 1
1 3 2
1 2 3
2 1 3
2 3 1
```

ในกรณีที่ข้อมูลที่ต้องการนำมาเรียงสับเปลี่ยนมีข้อมูลที่ซ้ำกัน เช่น $S = \{1,2,1\}$ เราสามารถปรับขั้นตอนวิธีข้างต้นเพื่อป้องกันสร้าง permutation ที่ซ้ำกันได้ โดยการตรวจสอบว่าสมาชิกที่กำลังจะสลับนั้นซ้ำกับตัวที่เคยสลับมาก่อนหรือไม่ หากซ้ำก็จะไม่ทำการสลับ ทำให้ไม่ได้ผลลัพธ์ไปใส่ใน dept ถัดไป ส่งผลให้ไม่มีการเรียก recursive call จากผลลัพธ์ตัวนั้น



รูปที่ 2.3

รูปที่ 2.3 แสดงตัวอย่างของการทำงานของขั้นตอนวิธีเพื่อสร้างการเรียงสับเปลี่ยนทั้ง 3 รูปแบบจากเซต $S = \{1,2,1\}$

- ที่ D0 จะไม่มีการสลับ 1 ตัวแรกกับ 1 ตัวที่สาม เพราะ 1 ตัวที่สามซ้ำกับ 1 ตัวแรก ทำให้ไม่ได้ผลลัพธ์ไปใส่ใน D1
- ที่ D1 จะไม่มีการสลับ 1 ตัวที่สองกับ 1 ตัวที่สาม เพราะ 1 ตัวที่สามซ้ำกับ 1 ตัวที่สอง ทำให้ไม่ได้ผลลัพธ์ไปใส่ใน D2

จำนวน permutation ที่เป็นผลลัพธ์จากขั้นตอนวิธีนี้จึงมีน้อยกว่า $N!$ เมื่อ N เป็นจำนวนของข้อมูลที่นำมาเรียงสับเปลี่ยน อย่างไรก็ตามขั้นตอนวิธียังมี time complexity เป็น $\theta(NN!)$

โปรแกรมที่ 2.5 การสร้าง permutation จากข้อมูลที่มีข้อมูลซ้ำ โดยใช้ unordered_set ในการบันทึกข้อมูลที่เคย swap แล้ว (Time Complexity: $O(NN!)$)

```
#include<iostream>
#include<vector>          // vector
#include<unordered_set>   // unordered_set
#include<utility>         // swap()
#include<cmath>           // tgamma()
using namespace std;

// Globals
vector< vector <int> > perms;  // keeps permutation
// Forwards
void genPermutation(vector <int> &vec, int idx);
void printVector(const vector<int> &vec);
void print2DVector(const vector< vector <int> > &vec);

int main(){
    int N; cin >> N;
    vector<int> data;
    data.resize(N);
    perms.reserve((int) tgamma(N+1));  // tgamma(N+1) = N!
    for(auto &i : data)
        cin>>i;
    cout << "Input\n";
    printVector(data);
    genPermutation(data, 0);
    cout << "Permutation\n";
    print2DVector(perms);
    cout << endl;
    return 0;
}
void genPermutation(vector <int> &vec, int idx){
    if(idx == vec.size())
        perms.emplace_back(vec);
    else{
        unordered_set<int>  hashset;   // keeps track of duplicate
        for(int i=idx; i<vec.size(); i++){
            if(hashset.count(vec[i]) == 1)  // duplicate found
                continue;
            hashset.insert(vec[i]);
            swap(vec[i], vec[idx]);
            genPermutation(vec, idx+1);
            swap(vec[i], vec[idx]);
        }
    }
}

```

```
void printVector(const vector<int> &vec){
    for(auto &i : vec)
        cout<<i<< " ";
    cout << "\n";
}
void print2DVector(const vector< vector <int> > &vec){
    for(auto &r : vec){      // O(MN)
        for(auto &c : r)
            cout << c << " ";
        cout << "\n";
    }
    cout << "\n";
}
```

ผลลัพธ์

3

1 2 1

Input

1 2 1

Permutation

1 2 1

1 1 2

2 1 1

นอกจากการเขียนโปรแกรมตามขั้นตอนวิธีโดยตรงแล้ว STL ยังมีฟังก์ชันมาตรฐานที่สามารถใช้ในการสร้าง permutation ที่เรียกว่า next_permutation( ) ได้ ฟังก์ชันนี้สามารถใช้กับข้อมูลที่แตกต่างกันทั้งหมดหรือที่มีซ้ำกันก็ได้ อย่างไรก็ตามเงื่อนไขของการใช้ฟังก์ชัน next_permutation( ) ก็คือข้อมูลที่นำมาสร้าง permutation ต้องเรียงแบบ lexicographical ก่อน ซึ่งเราก็สามารถจัดการได้โดยการเรียงลำดับข้อมูล (sorting) ก่อนเรียกใช้ฟังก์ชัน next_permutation( ) และด้วยเงื่อนไขนี้เอง ทำให้ผลลัพธ์ของ permutation ที่ได้ก็จะเรียงแบบ lexicographical ด้วย ดังนั้นหาก permutation ที่ต้องการสร้างจำเป็นต้องรักษาลำดับเดิมของข้อมูลนำเข้า การใช้ฟังก์ชัน next_permutation( ) จึงไม่ใช่ทางเลือกที่เหมาะสม

วิธีนี้มี time complexity เป็น $\theta(NN!)$

โปรแกรมที่ 2.6 การสร้าง permutation โดยใช้ STL ฟังก์ชัน next_permutation()

(Time Complexity: $O(NN!)$)

```
#include<iostream>
#include<vector>             // vector
#include<algorithm>          // sort(), next_permutation()
#include<cmath>              // tgamma()
using namespace std;
// Globals
vector< vector <int> > perms;    // keeps permutation
// Forwards
void genPermutation(vector <int> &vec, int idx);
void printVector(const vector<int> &vec);
void print2DVector(const vector< vector <int> > &vec);
int main(){
    int N; cin >> N;
    vector<int> data;
    data.resize(N);
    perms.reserve((int) tgamma(N+1));   // tgamma(N+1) = N!
    for(auto &i : data)
        cin>>i;
    cout << "Input\n";
    printVector(data);
    genPermutation(data, 0);
    cout << "Permutation\n";
    print2DVector(perms);
    cout << endl;
    return 0;
}
void genPermutation(vector <int> &vec, int idx){
    sort(vec.begin(), vec.end());
    do{
        perms.emplace_back(vec);
    } while(next_permutation(vec.begin(), vec.end()));
}
void printVector(const vector<int> &vec){
    for(auto &i : vec)
        cout<<i<< " ";
    cout << "\n";
}
void print2DVector(const vector< vector <int> > &vec){
    for(auto &r : vec){       // O(MN)
        for(auto &c : r)
            cout << c << " ";
        cout << "\n";
    }
    cout << "\n";
}
```

ผลลัพธ์

3

3 2 1

```
Input
3 2 1
Permutation
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

โปรแกรมที่ 2.7 การสร้าง permutation ของตัวอักขระใน string โดยใช้ STL ฟังก์ชัน next_permutation( )

(Time Complexity: $O(NN!)$)

```
#include<iostream>
#include<vector>              // vector
#include<string>              // string
#include<algorithm>           // sort(), next_permutation()
#include<cmath>               // tgamma()
using namespace std;
// Globals
vector<string> perms;       // keeps all permutation
// Forwards
void genPermutation(string &str);
void printVector(const vector<string> &vec);

```

```
int main(){
    string input; cin >> input;
    int N = input.size();
    perms.reserve((int) tgamma(N+1));  // tgamma(N+1) = N!
    cout << "Input\n";
    cout << input << "\n";
    genPermutation(input);
    cout << "Permutation\n";
    printVector(perms);
    cout <<  endl;
    return 0;
}
void genPermutation(string &str){
    sort(str.begin(), str.end());
    do{
        perms.emplace_back(str);
    } while(next_permutation(str.begin(), str.end()));
}
void printVector(const vector<string> &vec){
    for(auto &i : vec)
        cout<<i<< " ";
    cout << "\n";
}
```

ผลลัพธ์

ABCA

```
Input

ABCA

Permutation

AABC AACB ABAC ABCA ACAB ACBA BAAC BACA BCAA CAAB CABA CBAA
```

การจัดหมู่ (Combination)

การจัดหมู่ (Combination)

การจัดหมู่ของเซต (ที่มีสมาชิกที่แตกต่างกัน) คือการจัดกลุ่มของสมาชิกที่<u>แตกต่างกัน</u>ซึ่งเป็นสมาชิกของเซตนั้น

ตัวอย่างที่ 2.10 กำหนดเซต $S=\{a,b,c\}$

การจัดหมู่ทีละ 2 ของเซต S (2-combination of S) มีได้ 3 รูปแบบคือ

(1) a,b (2) a,c (3) b,c

กำหนดเซต $A=\{1,2,3,4\}$

การจัดหมู่ทีละ 2 ของเซต A (2-combination of A) มีได้ 6 รูปแบบคือ

(1) 1,2 (2) 1,3 (3) 1,4
(4) 2,3 (5) 2,4
(6) 3,4

การจัดหมู่ทีละ 3 ของเซต A (3-combination of A) มีได้ 4 รูปแบบคือ

(1) 1,2,3 (2) 1,2,4 (3) 1,3,4
(4) 2,3,4

ทฤษฎี 2 (Theorem 2)

ถ้า n เป็นจำนวนเต็มบวก และ r เป็นจำนวนเต็มโดยที่ $0 \le r \le n$

แล้ว มีจำนวนการจัดหมู่ทีละ r (r-combination) ของเซต (ที่มีสมาชิกไม่ซ้ำกัน) n ตัวเท่ากับ

$C(n,r)=\dfrac{P(n,r)}{r!}=\dfrac{n!}{r!(n-r)!}$, ($C(n,r)$ มีอีกชื่อคือสัมประสิทธิ์ทวิภาค หรือ binomial coefficient)

จากทฤษฎี 2 จะได้ว่า $C(n,n)=\dfrac{n!}{n!(n-n)!}=\dfrac{n!}{n!0!}=\dfrac{n!}{n!}=1$

บทเทียบ 2 (Corollary 2)

ถ้า n เป็นจำนวนเต็มบวก และ r เป็นจำนวนเต็มโดยที่ $0 \le r \le n$

แล้ว $C(n,r) = C(n,n-r)$

ตัวอย่างที่ 2.11 จงหาจำนวนของบิทสตริงความยาว 8 ตัวอักขระที่มีเลข 0 เป็นจำนวน 4 ตัว

คำตอบ ใช้ความรู้เรื่องการจัดหมู่ โดยเลือกตำแหน่ง 4 ตำแหน่งจาก 8 ตำแหน่ง และถ้าตำแหน่งใดถูกเลือก จะกำหนดให้ค่าที่ตำแหน่งนั้นเป็น 0 ในขณะที่ตำแหน่งใดที่ไม่ถูกเลือก จะกำหนดให้ค่าที่ตำแหน่งนั้นเป็น 1

ดังนั้นจำนวนของบิทสตริงความยาว 8 ตัวอักขระที่มีเลข 0 เป็นจำนวน 4 ตัว จึงเท่ากับจำนวนการจัดหมู่ทีละ 4 ของเซตที่มีสมาชิก 8 ตัว

$$C(8,4) = \frac{8!}{4! \times (8-4)!} = \frac{8 \times 7 \times 6 \times 5 \times 4!}{(4 \times 3 \times 2 \times 1) \times 4!} = 70$$

จำนวนบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 4 ตัว คือ 70 ตัว

ตัวอย่างที่ 2.12 จงหาจำนวนการจัดเรียงตัวอักขระที่อยู่ในสตริง ABABAAAB ในทุกรูปแบบ

คำตอบ ใช้ความรู้เรื่องการจัดหมู่ โดยเลือกตำแหน่ง 3 ตำแหน่งจาก 8 ตำแหน่ง และถ้าตำแหน่งใดถูกเลือก จะกำหนดให้ค่าที่ตำแหน่งนั้นเป็น B ในขณะที่ตำแหน่งใดที่ไม่ถูกเลือก จะกำหนดให้ค่าที่ตำแหน่งนั้นเป็น A

ดังนั้นจำนวนการจัดเรียงตัวอักขระที่อยู่ในสตริง ABABAAAB จึงเท่ากับจำนวนการจัดหมู่ทีละ 3 ของเซตที่มีสมาชิก 8 ตัว

$$C(8,3) = \frac{8!}{3! \times (8-3)!} = \frac{8 \times 7 \times 6 \times 5!}{(3 \times 2 \times 1) \times 5!} = 56$$

จำนวนการจัดเรียงตัวอักขระที่อยู่ในสตริง ABABAAAB คือ 56 แบบ

ข้อนี้สามารถพิจารณาเลือกตำแหน่ง 5 ตำแหน่งจาก 8 ตำแหน่ง และถ้าตำแหน่งใดถูกเลือก จะกำหนดให้ค่าที่ตำแหน่งนั้นเป็น A ในขณะที่ตำแหน่งใดที่ไม่ถูกเลือก จะกำหนดให้ค่าที่ตำแหน่งนั้นเป็น B ก็จะคำตอบเหมือนกัน

ตัวอย่างที่ 2.13 จงหาจำนวนบิทสตริงความยาว 8 ตัวอักขระที่มีเลข 0 เป็นจำนวน 3 หรือ 4 ตัว

คำตอบ ใช้กฎการบวกและความรู้เรื่องการจัดหมู่

ใช้กฎการบวกโดยพิจารณา 2 วิธีในการสร้างบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 3 หรือ 4 ตัว คือ

วิธีที่ 1 สร้างบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 3 ตัว จำนวนบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 3 ตัว เท่ากับ จำนวน**การจัดหมู่**ทีละ 3 ของเซตที่มีสมาชิก 8 ตัว คือ

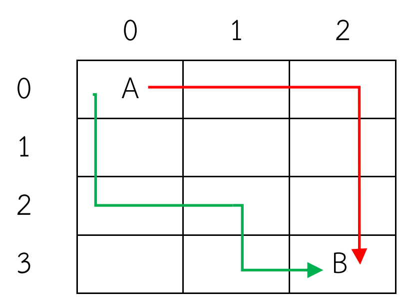
$$C(8,3)=\frac{8!}{3!\times(8-3)!}=\frac{8\times7\times6\times5!}{(3\times2\times1)\times5!}=56$$

วิธีที่ 2 สร้างบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 4 ตัว จำนวนบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 4 ตัว เท่ากับ คือ 70 (จากตัวอย่าง 2.11)

ใช้กฎการบวก ทำให้ได้จำนวนบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 3 หรือ 4 ตัว เป็น $56+70=126$ ตัว

สังเกตว่าไม่มีสมาชิกของเซตของบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 3 ตัว ที่ซ้ำกับสมาชิกของเซตของบิทสตริงความยาว 8 ที่มีเลข 0 เป็นจำนวน 4 ตัว ทำให้สามารถใช้กฎการบวกได้

ตัวอย่างที่ 2.14 ให้ตารางขนาด 4×3 จงหาจำนวนเส้นทางที่สั้นที่สุดของการเดินทางจากตำแหน่ง A ไปยังตำแหน่ง B ถ้ากำหนดให้เดินทางได้ 4 ทิศคือ ขึ้น (U), ลง (D), ซ้าย (L) และ ขวา (R) เท่านั้น



ตัวอย่าง เส้นทาง RRDDD คือเดินตามช่องต่อไปนี้ ตามลำดับ (0,0) (0,1) (0,2) (1,2) (2,2) (3,2)

เส้นทาง DDRDR คือเดินตามช่องต่อไปนี้ ตามลำดับ (0,0) (1,0) (2,0) (2,1) (3,1) (3,2)

คำตอบ เพื่อให้เดินด้วยระยะทางสั้นที่สุด ทิศทางที่จะเลือกมีเพียง D และ R เท่านั้น และจากขนาดของตาราง การเดินในทิศ D ขั้นต่ำคือ 3 ครั้ง และในทิศ R ขั้นต่ำคือ 2 ครั้ง

จำนวนเส้นทางที่เป็นไปได้เท่ากับจำนวนการจัดเรียงตัวอักขระที่อยู่ในสตริง DDDRR ที่เป็นไปได้ ซึ่งก็คือจำนวนการจัดหมู่ทีละ 2 ของเซตที่มีสมาชิก 5 ตัว (เลือกตำแหน่ง 2 ตำแหน่งจาก 5 ตำแหน่ง และถ้าตำแหน่งใดถูกเลือก จะกำหนดให้ค่าที่ตำแหน่งนั้นเป็น R ในขณะที่ตำแหน่งใดที่ไม่ถูกเลือก จะกำหนดให้ค่าที่ตำแหน่งนั้นเป็น D)

ดังนั้น จำนวนเส้นทางที่สั้นที่สุดจากตำแหน่ง A ไปยังตำแหน่ง B คือ $C(5,2)=\dfrac{5!}{2!\times 3!}=10$ เส้นทาง

โปรแกรมที่ 2.8 โปรแกรมคำนวณ $C(n,r)$ ด้วยทฤษฎี 2 โดยใช้ lgamma( )

```
#include<iostream>
#include<cmath>    //lgamma( )
using namespace std;

//Forwards
double computeCnrGamma(const int &n, const int &r);

int main(){
    int n,r; cin >> n >> r;
    long long cnr = (long long) computeCnrGamma(n,r);
    cout << cnr << endl;
    return 0;
}
double computeCnrGamma(const int &n, const int &r){
    double p = lgamma(n+1.0) - (lgamma(n-r+1.0)+ lgamma(r+1.0));
    return round(exp(p));
}
```

ผลลัพธ์

50 8

536878650

โปรแกรม 2.8 คำนวณ $C(n,r)$ โดยเลือกใช้ lgamma( ) เนื่องจากถ้าใช้ tgamma( ) ในการคำนวณฟังก์ชัน factorial อาจมีปัญหาจากตัวหาร $r!(n-r)!$ ซึ่งอาจมีค่ามาก (ผลลัพธ์เป็น INF) ทำให้ไม่สามารถทำการหารได้ ดังนั้นจึงเลือกใช้ lgamma( ) ที่คำนวณฟังก์ชันลอการิทึมแทน

โปรแกรมที่ 2.9 โปรแกรมคำนวณ $C(n,r)$ ด้วยทฤษฎี 2 (Time Complexity: $O(r)$)

```
1.   #include<iostream>
2.   using namespace std;
3.   //Forwards
4.   long long computeCnrMul(const int &n, const int &r);
5.
6.   int main(){
7.       int n,r; cin >> n >> r;
8.       long long cnr = computeCnrMul(n,r);
9.       cout << cnr << endl;
10.      return 0;
11.  }
12.  long long computeCnrMul(const int &n, const int &r){
13.      if (r == 0)
14.          return 1;
15.      else if (r > n / 2)
16.          return computeCnrMul(n, n - r); // Corollary 2
17.      else{
18.          long long res = 1;
19.          for (int k = 1; k <= r; ++k){
20.              res *= n - k + 1;
21.              res /= k;
22.          }
23.          return res;
24.      }
25.  }
```

ผลลัพธ์

```
30 18
86493225
```

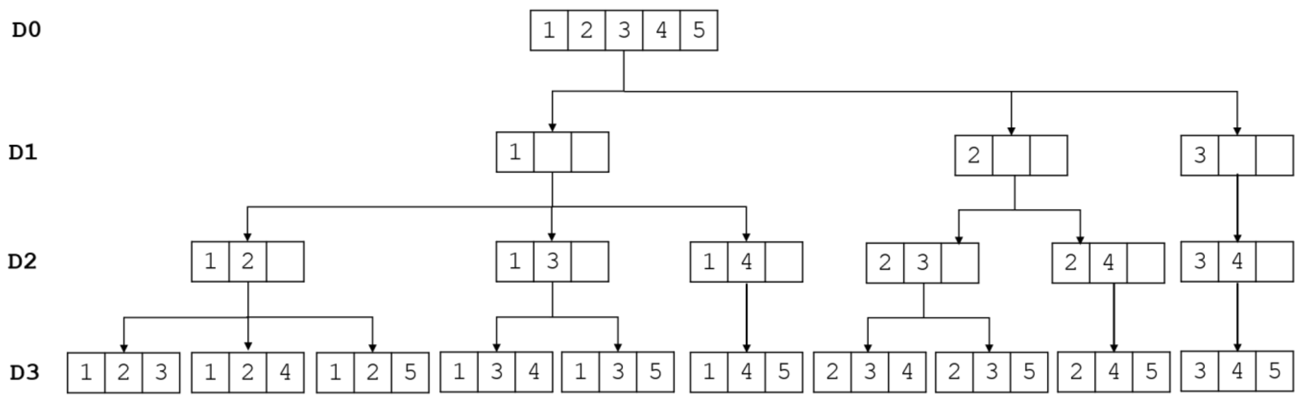
โปรแกรม 2.9 คำนวณ $C(n,r)$ โดยใช้การหาผลคูณสะสม ที่ได้จากการจัดรูปของสมการต่อไปนี้

$$
\begin{aligned}
C(n,r) = \frac{n!}{r!(n-r)!} &= \frac{n(n-1)...(n-r+1)}{1\times 2\times 3\times ...\times r} \\
&= \left(\frac{n}{1}\right)\left(\frac{n-1}{2}\right)\left(\frac{n-2}{3}\right)...\left(\frac{n-r+1}{r}\right) \\
&= \prod_{k=1}^{r}\left(\frac{n-k+1}{k}\right)
\end{aligned}
$$

บรรทัดที่ 16 ใช้บทเทียบ 2 เพื่อเลือกคำนวณกรณีที่ r มีค่าน้อยที่สุด ทำให้ลดจำนวนรอบในการวน loop ที่บรรทัด 19-22

การสร้าง r-combination จากเซตที่มีสมาชิก N ตัว

เทคนิคที่ใช้ในการเขียนโปรแกรมเพื่อสร้าง (generate) การจัดหมู่ทีละ r ของเซตที่มีสมาชิกที่แตกต่างกัน N ตัว (r-combination) ก็คือ backtracking เช่นเดียวกับการสร้างการเรียงสับเปลี่ยน (permutation) นั่นเอง



รูปที่ 2.4

รูปที่ 2.4 แสดงตัวอย่างของการทำงานของขั้นตอนวิธีเพื่อสร้างการจัดหมู่ทีละ 3 ของเซต $S = \{1,2,3,4,5\}$

- ที่ D0 เป็นการเรียก recursive function ครั้งแรกด้วยสมาชิกทุกตัวในเซต โดยกำหนดตำแหน่งข้อมูลตัวแรกเป็น 1 และตัวสุดท้ายเป็น 5 ในขั้นนี้ต้องมีการสร้างอาร์เรย์หรือ vector ขนาด 3 ที่จะใช้เก็บข้อมูล 3-combination แต่ละรูปแบบ
- ที่ D1 จะนำสมาชิกตัวแรกถึงตัวที่ 3 มาเติมในตำแหน่งแรกของ r-combination จากนั้นทำการเรียก recursive call ใน dept D2 ต่อไป จำนวน recursive call ที่เรียกขึ้นอยู่กับจำนวนสมาชิกที่อยู่หลังสมาชิกที่นำมาเติมที่ตำแหน่งแรก ซึ่งต้องมีอย่างน้อย 2 หรือ $N-r$ ตัว
- ที่ D2 จะนำสมาชิกที่อยู่หลังสมาชิกตัวแรกของ r-combination มาเติมที่ตำแหน่งที่ 2 จากนั้นทำการเรียก recursive call ใน dept D3 ต่อไป จำนวน recursive call ที่เรียกขึ้นอยู่กับจำนวนสมาชิกที่อยู่หลังสมาชิกทตัวที่ 2 ที่นำมาเติมใน r-combination ซึ่งจะต้องมีอย่างน้อย 1 หรือ $N-r-1$ ตัว

- ที่ D3 จะนำสมาชิกที่อยู่หลังสมาชิกตัวที่ 2 ของ r-combination มาเติมที่ตำแหน่งที่ 3 จากนั้นทำการเรียก recursive call ต่อไป แต่จะพบกับ base case เพราะได้เติมทุกตำแหน่งของ r-combination จนเต็มแล้ว recursion จะหยุดการทำงานและคนค่า r-combination มาเป็นผลลัพธ์

Time complexity ของขั้นตอนวิธีนี้ คำนวณได้จาก จำนวนของ r-combination $C(N,r)$ ซึ่งมี Big-O เป็น $O(N^r)$ คูณด้วย dept ของ recursive call ซึ่งมีซึ่งมี Big-O เป็น $O(N)$ ทำให้ได้ time complexity ของขั้นตอนวิธีเป็น $O(N^{r+1})$

โปรแกรมที่ 2.9 การสร้าง r-combination จากสมาชิกทุกตัวของเซตที่มีสมาชิกที่แตกต่าง

(Time Complexity: $O(N^{r+1})$)

```
#include<iostream>
#include<vector>          // vector
using namespace std;

// Globals
vector< vector <int> > combs;    // keeps combination
// Forwards
void genCombination(vector <int> &vec, vector <int> &rcomb,
                    int r, int idl, int idr, int idx);
void printVector(const vector<int> &vec);
void print2DVector(const vector< vector <int> > &vec);
long long computeCnrMul(const int &n, const int &r);

int main(){
    int N, R; cin >> N >> R;
    vector<int> data;
    data.resize(N);
    combs.reserve(computeCnrMul(N,R));
    for(auto &i : data)
        cin>>i;
    cout << "Input\n";
    printVector(data);
    vector<int> rcomb(R);
    genCombination(data,rcomb,R,0,N-1,0);
    cout << "Combination\n";
    print2DVector(combs);
    cout << endl;
    return 0;
}

```

```
void genCombination(vector <int> &vec, vector <int> &rcomb,
                    int r, int idl, int idr, int idx){
    if(idx == r)
        combs.emplace_back(rcomb);
    else{
        for(int i=idl; i<=idr && idr – idx + 1 >= r - idx; i++){
            rcomb[idx] = vec[i];
            genCombination(vec,rcomb,r,i+1,idr,idx+1);
        }
    }
}
void printVector(const vector<int> &vec){
    for(auto &i : vec)
        cout<<i<< " ";
    cout << "\n";
}
void print2DVector(const vector< vector <int> > &vec){
    for(auto &r : vec){      // O(MN)
        for(auto &c : r)
            cout << c << " ";
        cout << "\n";
    }
    cout << "\n";
}
long long computeCnrMul(const int &n, const int &r){
    if (r == 0)
        return 1;
    else if (r > n / 2)
        return computeCnrMul(n, n - r); // Corollary 2
    else{
        long long res = 1;
        for (int k = 1; k <= r; ++k){
            res *= n - k + 1;
            res /= k;
        }
        return res;
    }
}
```

ผลลัพธ์

```
5 3
1 2 3 4 5
Input
1 2 3 4 5
Combination
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Miscellaneous

- เลือกแสดงผลบรรทัดใหม่ (new line) ด้วย “\n” ซึ่งเร็วกว่า endl ที่ต้องทำการ flush ostream ทุกครั้ง ยกเว้นบรรทัดก่อน return 0 ในฟังก์ชัน main( ) ที่ endl สามารถรับประกันว่าเมื่อจบการทำงานโปรแกรมส่งผลลัพธ์ออกไป
- การเข้าถึงสมาชิกของ vector แบบ random access ด้วย operator[ ] เร็วกว่าฟังก์ชัน at( ) แต่การเข้าถึงด้วย operator[ ] อาจไม่ปลอดภัย เพราะไม่ได้ตรวจสอบขอบเขตของ index
- หากทราบขนาดของ vector ควรเลือกที่จะสร้าง vector ตามขนาดที่จะใช้ แทนการใช้ฟังก์ชัน push_back( ) เพื่อป้องกัน reallocation ที่ไม่จำเป็นระหว่างการเพิ่มข้อมูล (โดยเฉพาะการเพิ่มข้อมูลจำนวนมาก) ทั้งนี้อาจสร้าง vector แบบไม่มีสมาชิก และทำการปรับขนาดครั้งเดียวด้วยฟังก์ชัน resize() หรือ reserve( ) ตามความเหมาะสม
- เพิ่มสมาชิกให้ vector ด้วยฟังก์ชัน emplace_back () และ emplace( ) แทน push_back( ) และ insert( ) เพื่อลดการสร้างและคัดลอก (copy) ข้อมูลที่เป็น parameter/argument ซ้ำซ้อน
- เลือกใช้การส่งค่า parameter ด้วย pass by reference (&) แทนการ pass by value ทุกครั้งที่ใช้ได้ เพื่อลดหน่วยความจำที่โปรแกรมต้องใช้จากการคัดลอก (copy) ข้อมูลระหว่างการส่งค่า parameter
- ตัวแปรที่เก็บข้อมูลขนาดใหญ่ (> 4MB หรือประมาณ `int array[100000]`) ต้องประกาศเป็น global variable เพราะหน่วยความจำที่กำหนดให้แต่ละฟังก์ชันใช้มีจำกัด
- สำหรับเขียนโปรแกรมเพื่อการแข่งขัน เราสามารถ include header file ชื่อ bits/stdc++.h (#include <bits/stdc++.h>) ได้เพื่อความสะดวก แต่สำหรับการเขียนโปรแกรมเพื่อจุดประสงค์อื่นไม่ควร include header file แบบเหมารวมนี้

อ้างอิง

Kenneth H. Rosen, 2012: **Discrete Mathematics and its Applications 7th ed.**, McGraw-Hill Companies Inc., New York.